Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism*

Stephen W. Keckler and William J. Dally
Artificial Intelligence Laboratory and Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

The technology to implement a single-chip node composed of 4 high-performance floating-point ALUs will be available by 1995. This paper presents processor coupling, a mechanism for controlling multiple ALUs to exploit both instruction-level and inter-thread parallelism, by using compile time and runtime scheduling. The compiler statically schedules individual threads to discover available intra-thread instruction-level parallelism. The runtime scheduling mechanism interleaves threads, exploiting inter-thread parallelism to maintain high ALU utilization. ALUs are assigned to threads on a cycle by cycle basis, and several threads can be active concurrently. We provide simulation results demonstrating that, on four simple numerical benchmarks, processor coupling achieves better performance than purely statically scheduled or multi-processor machine organizations. We examine how performance is affected by restricted communication between ALUs and by long memory latencies. We also present an implementation and feasibility study of a processor coupled node.

1 Introduction

By 1995, improvements in semiconductor technology will allow multiple high performance floating point units and several megabits of memory to reside on a single chip. One possible use of these multiple arithmetic units is to organize them in a single processor to exploit instruction-level parallelism. Controlling concurrent function units presents a challenge. Applications exhibit an uneven amount of instruction-level parallelism during their execution [11]. In some parts of a program, all of the function units will be used, while in others, serial computations with little instruction-level parallelism dominate. The amount of available parallelism depends upon both the computation at hand and the accessibility of data. Long memory latencies can stifle the opportunities to exploit instruction-level parallelism. The ideal multiple function unit processing node should provide a task the use of as many function units as it needs, but also allow unused units to be assigned to other tasks. In addition to being effective as a uniprocessor, such a node serves well in a parallel computing environment since many tasks are available. Furthermore, exploiting instruction-level parallelism in conjunction with coarser grained algorithmic concurrency improves machine performance.

*The research described in this paper was supported in part by a National Defense Science and Engineering Graduate Fellowship, by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825, and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation, and AT&T.

Appears in the 19th Annual International Symposium on Computer Architecture

Processor Coupling

Processor coupling is a runtime scheduling mechanism in which multiple function units execute operations from multiple instruction streams and place results directly in each other's register files. Several threads may be active simultaneously sharing use of the function unit pipelines. Instruction level parallelism within a single thread is exploited using static scheduling techniques similar to those demonstrated in the Multiflow Trace system [4]. At runtime, the hardware scheduling mechanism interleaves several threads exploiting inter-thread parallelism to maintain high utilization of function units.

Figure 1 demonstrates how processor coupling dynamically interleaves instruction streams from multiple threads across multiple function units. The operations from threads **A**, **B**, and **C** are scheduled independently at compile time as shown in the top of the figure. Each column in a thread's instruction stream represents an operation field for a single function unit. Each row holds operations that may be executed simultaneously. The empty boxes indicate that there is insufficient instruction-level parallelism to keep all of the function units busy. During execution, arbitration for function unit usage is performed on a cycle by cycle basis. When several threads are competing for a given function unit, one is granted use and the others must wait. For example, operations **A3** and **A4** are locked out during the second cycle because thread **C** is granted those units instead. Figure 2 illustrates the mapping of function units to threads as a result of runtime arbitration for the first two cycles shown in Figure 1.

Note that operations scheduled in a single long instruction word need not be executed simultaneously. Allowing the static schedule to slip provides for finer grain sharing of function units between threads. In Figure 1, operations **A3**-**A6** are scheduled in the same instruction word for thread **A**. Operations **A3**, **A5**, and **A6** are all

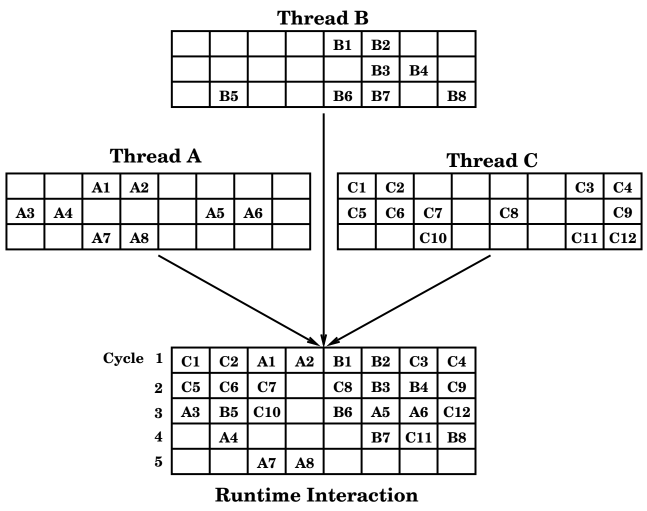

Figure 1: Interleaving of instruction streams: threads A, B, and C are scheduled separately and their instruction streams are shown at the top of the diagram. Each column of the instruction stream is a field for a particular function unit and each row holds those operations which are allowed to execute simultaneously. The bottom box shows a runtime interleaving of these threads in which some operations are delayed due to function unit conflicts.

issued during cycle 3 while **A4** is not issued until cycle 4. However, **A4** must be issued before **A7** and **A8**.

A compiler can be used to extract the maximum amount of statically available instruction-level parallelism from a program fragment. However, compile time scheduling is limited by unpredictable memory latencies and by some dependencies, such as data dependent array references, which cannot be statically determined. Furthermore, although trace scheduling [5] and software pipelining techniques [14] can be used, branch boundaries tend to limit the number of operations that can be scheduled simultaneously. By interleaving multiple threads, the hardware runtime scheduling mechanisms of processor coupling address the limits of static scheduling due to dynamic program behavior. Additional information and further analysis of processor coupling can be found in [12].

Related Work

Processor coupling incorporates ideas from research in compile time scheduling, multiple instruction issue architectures, multithreaded machines, and runtime scheduling. Very long instruction word (VLIW) processors such as the Multiflow Trace series [3] use only compile time scheduling to manage instruction-level parallelism and resource use. Superscalar processors execute multiple instructions simultaneously by relying upon runtime scheduling mechanisms to determine data dependencies [18, 10]. The proposed XIMD [19] architecture employs compile time techniques to statically schedule instructions as well as threads, but does not dynamically interleave multiple thread execution. Using multiple threads to hide memory latencies and pipeline delays has been explored in [7, 8, 13, 17]. Dataflow and hybrid dataflow approaches such as [2, 9] have decomposed programs into large numbers of threads consisting of one or a few instructions which are dynamically scheduled. Fisher and Rau summarize many instruction-level parallelism techniques in [6].

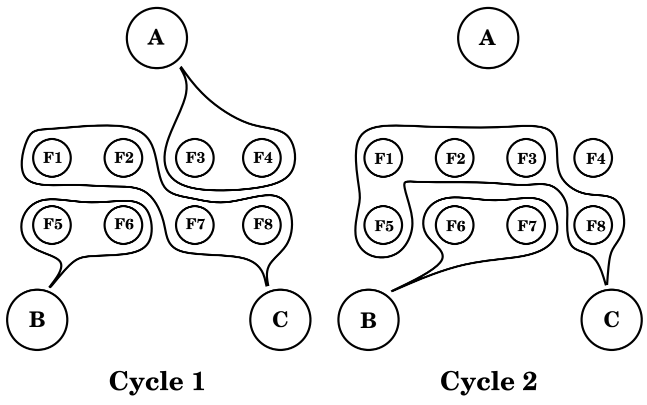

Figure 2: Two mappings of function units to threads. These mappings correspond to the first two cycles shown in Figure 1.

Context of Processor Coupling

Processor coupling is useful in machines ranging from workstations based upon a single multi-ALU node to massively parallel machines such as the MIT M-Machine, which is currently being designed. The M-Machine will consist of thousands of multi-ALU processor coupled nodes and will have many threads to be interleaved at each node. The machine will thus take advantage of a hierarchy of parallelism, ranging from coarse-grained algorithmic parallelism to extremely fine-grained instruction-level parallelism. However, as will be demonstrated in Section 4, processor coupling can be effective on only a single node. This paper will consider only a single node instance of processor coupling.

Outline

The next section examines in more detail the architecture of a processor coupled node. Section 3 presents a prototype compiler for processor coupling, as well as a simulator to evaluate processor coupled performance. Experimental assumptions, benchmarks, and results are described in Section 4. Section 5 sketches an implementation of processor coupling while Section 6 shows that building a processor coupled node will be feasible. Finally, Section 7 proposes further directions for research in this area.

2 Discussion of Architecture

A processor coupled node, as shown in Figure 3 consists of a collection of function units, register files, memory banks, and interconnection networks. A function unit may perform integer operations, floating point operations, branch operations, or memory accesses. Function units are grouped into *clusters* sharing a register file among them. A cluster can write to its own register file or to that of another cluster through the Unit Interconnection Network.

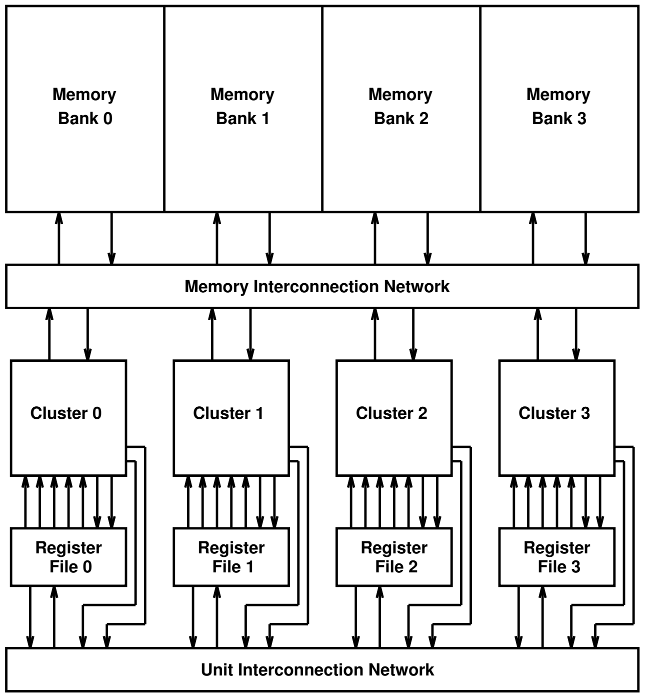

Figure 3: This sample machine consists of four clusters, each of which contains an arithmetic unit and a memory unit. The units communicate with each other through the unit interconnection network and through memory.

Clusters access memory banks through the Memory Interconnection Network.

A function unit may be generic, such as an integer ALU, or more specialized, such as a branch calculation unit or a floating point multiplier; a unit may be pipelined to arbitrary depth. A thread's instruction stream can be considered to be a sparse matrix of *operations*, as seen in Figure 1. To keep signal propagation delays as short as possible, control is distributed throughout the clusters. Each function unit contains an *operation cache* and an *operation buffer*. When summed over all function units, the operation caches form the instruction cache. The operation buffer holds a pending operation from each active thread. A cluster's register file is multi-ported to allow multiple read and write operations per cycle. Although execution of a thread's instruction does not take place in lock step, global management of an instruction pointer prevents operations from issuing out of order. Section 5 discusses the architecture of a function unit in more detail.

Intra-thread Synchronization

Processor coupling uses data presence bits in registers for low level synchronization within a thread. An operation will not be issued until all of its source registers are valid. When an operation is issued, the valid bit for its destination register is cleared. The valid bit is set when the operation completes and writes data back to the register file. To move data between function units, an operation may specify destination registers in other clusters. Thus, registers are used to indicate data dependencies between individual operations and to prevent operations from executing before their data requirements are satisfied.

An operation can issue only if all of its data dependencies are satisfied and all of the previous instruction's operations have issued. Because different function units may have different pipeline latencies, this discipline ensures in-order operation issue, but not necessarily in-order completion.

Multiple Threads

Hardware is provided to sequence and synchronize a small number of active threads. Each thread has its own instruction pointer and logical set of registers, but shares the function units and interconnect bandwidth. A thread's register set is distributed over all of the clusters that it uses. The combined register set in each cluster can be implemented as separate register files or as a collection of virtually mapped registers [15]. Communication between threads takes place through the shared memory on the node; synchronization between threads is on the presence or absence of data in a memory location.

Each function unit determines independently, through examination of dynamic data dependencies, the next operation to issue. That operation may be from any thread in the active set. The function unit examines simultaneously the data requirements for each pending operation by inspecting the valid bits in the corresponding register files. The unit selects a ready operation, marks its destination register invalid, and issues it to the execution pipeline stages.

A processor coupled system must provide a set of thread management functions. If a thread in the active set idles, it may be swapped out in favor of another thread waiting to execute. The process of spawning new threads and of terminating threads must occur with low latency as well. Thread management is beyond the scope of this paper and will not be further discussed.

Memory System

The memory system is used for storage, synchronization, and communication between threads. Like the registers, each memory location has a valid bit. Different flavors of loads and stores are used to access memory locations. The capabilities of memory reference operations are similar to those in the Tera machine description [1] and are summarized in Table 1. These mechanisms can be used to build producer-consumer relationships, atomic updates, semaphores, and other types of synchronization schemes.

Reference	Precondition	Postcondition
load	unconditional	leave as is
	wait until full	leave full
	wait until full	set empty
store	unconditional	set full
	wait until full	leave full
	wait until empty	set full

Table 1: Loads and stores can complete if the addressed location's valid bit satisfies the reference's precondition. When a memory reference completes, it sets the valid bit to the specified postcondition.

On-chip memory is used as a cache and is interleaved into banks to allow concurrent access to multiple memory locations. Memory operations that must wait for synchronization are held in the memory system. When a subsequent reference changes a location's valid bit, waiting operations reactivate and complete. This split transaction protocol reduces memory traffic and allows memory units to issue other operations.

3 Experimental Environment

In order to evaluate the performance of processor coupling, we built an experimental environment consisting of a compiler and a simulator. The compiler is implemented in Common Lisp and the simulator in C++.

Compiler

The compiler source language has simplified C semantics with Lisp syntax. A configuration file for the machine to be simulated specifies the number and type of function units, each function unit's pipeline latency, and the grouping of function units into clusters. The compiler uses configuration information to statically schedule thread operations.

Partitioning of a benchmark program into multiple threads is made explicit in the source code using `fork` and `forall` constructs. A `fork` causes the enclosed expression to be compiled separately, and at runtime the forked thread runs concurrently with the host thread.

Code can be compiled in two ways depending on the value of the `mode` flag. If set to `single`, each thread's code is scheduled on the function units of a single cluster. The compiler chooses upon which cluster a given thread will be scheduled. If set to `unrestricted`, each thread may use as many of the function units as it needs. The compiler assigns an ordered list of clusters to each thread, which determines the function unit access pattern. Using different orderings for different threads serves as a simple form of load balancing.

The compiler performs several optimizations including constant propagation, common subexpression elimination, and static evaluation of expressions with constant operands. Live variables are kept in registers across basic block boundaries. The compiler does not perform register allocation, assuming that an infinite number of registers are available. Simulation results show that the realistic machine configurations all have a peak of fewer than 60 live registers per cluster, for each of the selected benchmarks. Averaging over these benchmarks, each cluster uses a peak of 27 registers. Only ideal mode simulations, in which loops are unrolled extensively by hand, require as many as 490 registers.

Scheduling is done according to critical path analysis of each basic block in which the most critical operations are scheduled first. Operations are placed to minimize the amount of communication between function units.

The compiler does not perform trace scheduling or software pipelining, and does not schedule or move code across basic block boundaries. Loops must be unrolled by hand and procedures are implemented as macro-expansions. Although a few modern compilers do perform trace scheduling and software pipelining, this compiler provides a good lower bound on the quality of generated code. Using more sophisticated scheduling techniques should benefit processor coupling at least as much other machine organizations.

The compiler produces assembly code, a diagnostic file, and a modified configuration file with information concerning register and memory requirements.

Simulator

Like the compiler, the simulator is parameterized for processor configurations. The simulator takes the configuration file and the assembly code produced by the compiler, runs the program, and generates statistics including dynamic cycle count, operation count, and function unit utilization.

Simulation is at a functional level rather than at a register transfer level, but the simulator is accurate in counting the number of cycles and operations executed. Certain assumptions were made in order to simplify simulation. Each thread is allocated its own set of registers while the function units, the communication channels, and the memory system are shared. No thread management is considered as all executing threads are assumed to be a part of the active set. New threads are spawned by the `fork` instruction which initiates a new thread context. The number of available registers is specified by the compiler, and integers and floating point numbers reside in the same register files.

The memory system is modeled statistically. The configuration file specifies the hit latency, the miss rate, and a minimum and maximum miss penalty. If a miss occurs, the number of penalty cycles is randomly chosen from the penalty range. Also, no bank conflicts are modeled as a memory operation can always access the necessary bank. No instruction cache misses or operation prefetch delays are included.

Additionally, the configuration file describes the behavior of the interconnection between function units. The simulator manages arbitration for buses between function units if conflicts arise.

Simulation Modes

Since an input program specifies the amount of threading, and the compiler can adjust how operations are assigned to clusters, five different modes of operation are possible. Each mode corresponds to a different type of machine and is described below:

1. Sequential (SEQ): The program is written using only a single thread with the compiler scheduling the operations on only one cluster. This is similar to a statically scheduled machine with an integer unit, a floating point unit, a memory unit, and a branch unit.

2. Statically Scheduled (STS): Like Sequential mode, only a single thread is used, but there is no restriction on the clusters used. This approximates a VLIW machine without extensive trace scheduling.

3. Ideal: The program is single threaded, has its loops unrolled as much as possible, and is completely statically scheduled. This mode is not available for those benchmarks with data dependent control structures as they cannot be statically scheduled.

4. Thread per element (TPE): Multiple threads are specified, but each thread is restricted to run on only one cluster. Static load balancing is performed to schedule different threads on different clusters. A thread may not migrate to other clusters, but the benchmark programs are written to divide work evenly among the clusters.

5. Coupled: Multiple threads are allowed and function unit use is not restricted.

4 Experiments

Benchmarks

We selected four simple benchmarks to test the effectiveness of processor coupling. The benchmark suite includes applications that exhibit both data parallelism and data dependent control parallelism. Each benchmark is written so that it can be partitioned easily into separate threads. This is intended to provide a set of benchmarks that each machine model can perform upon well.

Matrix is a 9×9 matrix multiply of floating point numbers, with the inner loop unrolled completely. The threaded versions execute all of the iterations of the outer loop in parallel. The ideal version of this benchmark has all of the loops unrolled and the entire computation is statically scheduled by the compiler.

FFT is a 32 point decimation-in-time fast-Fourier-transform of complex numbers. A sequential data movement routine places the input vector in bit-flipped order. The threaded version executes concurrently all of the butterfly computations within a single stage. The ideal version unrolls the inner loop completely statically scheduling all of the operations within a single stage.

LUD solves a sparse system of linear equations using the lower-upper decomposition technique. The input data is a 64×64 adjacency matrix of an 8×8 mesh. After selecting a source row, the threaded version updates all of the target rows concurrently. No loops are unrolled and there is no ideal version since the control flow depends upon the input data.

Model is a model evaluator from an VLSI circuit simulator in which the change in current for each node in the network is computed based upon previous node voltages. The input circuit is a 20 device CMOS operational amplifier. The threaded version creates a new thread to evaluate each device on each iteration of a master loop. Like **LUD**, no loops are unrolled.

Although they are small and well contained problems, these benchmark programs can be used as building blocks for larger numerical applications. For example, the compute intensive portions of a circuit simulator such as SPICE include a model evaluator and sparse matrix solver [16].

Using these benchmarks, we first compare the performance of the different types of simulated machines. Further experiments explore issues concerning restricted connectivity between function units, variable memory latencies, and different mixes of function units.

Baseline Comparisons

The baseline machine consists of four arithmetic clusters and two branch clusters. Each arithmetic cluster contains an integer unit, a floating point unit, a memory unit, and a shared register file, while a branch cluster contains only a branch unit and a register file. The branch cluster may be used by any thread for all modes of simulation. Although processor coupling does not preclude multiple branch units, since the current version of the compiler allows each thread to issue at most one branch operation per cycle, one branch cluster is sufficient. Each function unit has a pipeline latency of one cycle.

			Compared	Utilization	
Benchmark	Mode	#Cycles	to Coupled	FPU	IU
Matrix	SEQ	1992	3.12	0.69	0.90
Matrix	STS	1182	1.85	1.16	1.52
Matrix	TPE	629	0.99	2.19	2.83
Matrix	Coupled	638	1.00	2.16	2.79
Matrix	Ideal	350	0.55	3.93	0.28
FFT	SEQ	3377	3.06	0.24	0.61
FFT	STS	1792	1.63	0.45	1.24
FFT	TPE	1977	1.79	0.40	1.05
FFT	Coupled	1102	1.00	0.73	2.03
FFT	Ideal	402	0.36	1.99	2.54
Model	SEQ	993	2.69	0.21	0.10
Model	STS	771	2.09	0.27	0.13
Model	TPE	395	1.07	0.54	0.64
Model	Coupled	369	1.00	0.57	0.70
LUD	SEQ	57975	2.69	0.14	0.45
LUD	STS	33126	1.54	0.24	0.78
LUD	TPE	22627	1.05	0.35	1.35
LUD	Coupled	21543	1.00	0.37	1.42

Table 2: Cycle count comparison of different types of machines. Floating point unit (FPU) utilization is given as the average number of floating point operations executed each cycle. Integer Unit (IU) utilization is calculated similarly.

In the baseline machine, an operation can specify at most two simultaneous register destinations. A function unit can write a result back to any cluster's register file, each of which has enough buses and ports to prevent resource conflicts. Memory units perform the operations required for address calculation. Memory references take a single cycle and have no bank conflicts.

Assuming that the compiler produces the best possible schedule, the number of cycles executed by the fully unrolled instance of a benchmark is a lower bound for these particular hardware resources. Sequential mode operation provides an upper bound since only the parallelism within a single cluster can be exploited. Table 2 shows the cycle counts for each machine mode. Floating point utilization is given as the average number of floating point operations executed each cycle. Integer unit utilization is calculated similarly. Figure 4 displays the cycle counts graphically.

Considering the statically scheduled benchmarks, STS mode requires on average 1.7 times fewer cycles than SEQ, since STS allows use of all the function units. Because the Ideal mode's program is fully unrolled and statically scheduled, loop overhead operations and redundant address calculations are eliminated. This reduced operation count permits the ideal machine to execute in an average of 7 times fewer cycles than SEQ.

In threaded mode, the cycle count for Coupled and TPE are nearly equivalent for the **Matrix**, **LUD**, and **Model** benchmarks which have been stripped of nearly all sequential execution sections, and are easily partitionable. TPE is as fast as Coupled since the

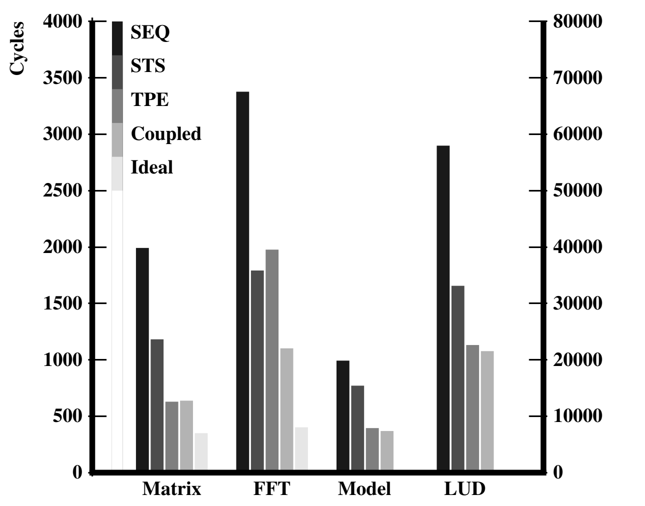

Figure 4: Baseline cycle counts for the five simulation modes. **Matrix**, **FFT**, and **Model**, are referenced to the left axis, while the scale for **LUD** is on the right. Ideal is only implemented for **Matrix** and **FFT**.

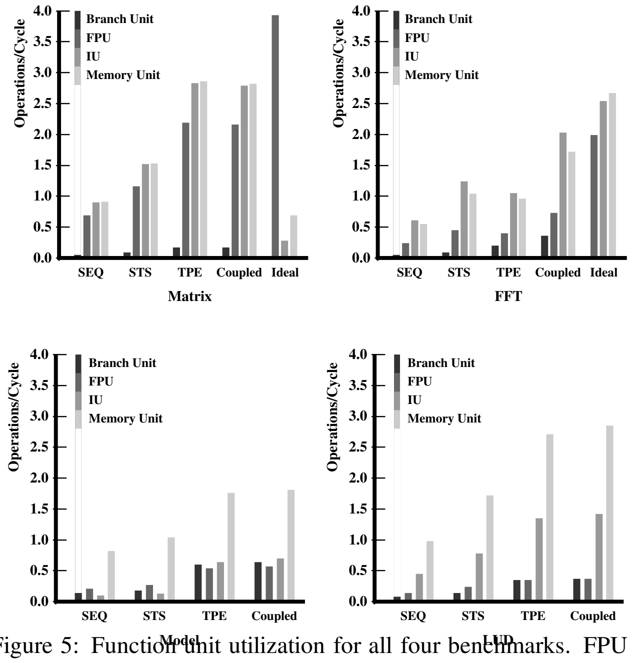

Figure 5: Function unit utilization for all four benchmarks. FPU utilization is the average number of FPU operations per cycle. Other unit utilizations are calculated similarly.

load is balanced across its clusters. Processor coupling will have an advantage in less intrinsically load balanced computations as long as threads are not allowed to migrate between clusters.

FFT, however, has a large sequential section that cannot be partitioned. Since each TPE thread can execute on only one cluster, the performance of TPE in the sequential section is no better than SEQ. In fact, because of the dominance of the sequential section, TPE does not even perform as well as STS for the entire benchmark. Coupled, on the other hand, performs as well as STS on sequential code. The available instruction level parallelism can be exploited by Coupled, but not by TPE. Thus, one advantage of Coupled over TPE is found in sequential code execution. Since parallel speedup of a program is limited by the amount of sequential code, single thread performance is vital.

The dynamic scheduling capability of Coupled mode allows it to execute in an average of 55% fewer cycles than the statically scheduled STS mode. TPE, with its multiple threads, also executes faster than STS for all benchmarks but **FFT**. This reduced cycle count in TPE and Coupled is due to the fine grained interleaving of threads. The multiple threads of Coupled mode result in much higher function unit utilization, and therefore a lower cycle count, than STS. One of processor coupling's advantages over a statically scheduled scheme is the increased unit utilization allowed by dynamic scheduling.

Utilization

Figure 5 shows for each benchmark, the utilization of the floating point units, integer units, memory units, and branch units. In all benchmarks, unit utilization increases as the simulation mode approaches Ideal. For **Matrix**, utilization is balanced for FPUs, IUs, and memory units, up to the Coupled mode. In Ideal mode, the FPU utilization is 3.9, indicating that the compiler has filled nearly every floating point operation slot. Note that since the compiler has eliminated loop overhead operations and common subexpressions in array index calculation, very few integer and branch operations are required. Furthermore, a significant fraction of the memory operations have been replaced by register operations.

Aside from the ideal mode, **FFT** utilization is similar. The multiple active threads available in TPE and Coupled modes drive memory unit utilization up. This rise results from the many read and write operations performed by the butterfly calculations of the inner loop. Memory utilization in the Ideal mode is still high because the compiler was unable to replace memory references with register references. The **Model** and **LUD** benchmarks are dominated by memory operations. Thus, even in Coupled and TPE mode, the integer and floating point utilizations are still quite small.

Interference

For multithreaded versions of the benchmarks programs, the statically scheduled threads interfere with one another, causing the runtime cycle count to be longer than the compile time schedule would suggest. To demonstrate this dilation, we use a slightly different version of the **Model** benchmark in Coupled mode. Four threads are created when the program starts. Each thread accesses a common priority queue of devices to be evaluated, chooses a device, updates the queue, and then evaluates the device. This loop is repeated by all threads until the queue is empty. A new input circuit with identical devices, each at the same operating point, allows extraneous code in the source program to be removed. Thus, every operation specified in the new source program is executed. This provides us with ability to compare runtime cycle count with the number of instructions generated by the compiler. With the

Mode	Thread	Compile Time Schedule	Runtime Cycle Count	Devices Evaluated
STS	1	25	25.0	20
Coupled	1	23	28.0	8
Coupled	2	23	38.7	6
Coupled	3	23	77.3	3
Coupled	4	23	80.7	3

Table 3: Average cycle counts for each iteration of the inner loop of the **Model** benchmark for STS and Coupled with threads assigned different priorities.

modified benchmark, the effect of the priority assigned to a thread on its runtime schedule can be seen more clearly. We compare the Coupled benchmark to a similarly altered version of an STS mode program.

Table 3 shows the compile time schedule length and the average runtime cycle count to evaluate one model for each of the four threads in Coupled mode. The higher priority threads in Coupled mode execute in fewer cycles. In STS mode, there is only one thread, and it runs in the same number of cycles as the static schedule predicts.

In addition, contention between threads for the shared queue arises in Coupled mode as even the highest priority thread requires more cycles than the schedule predicts. Taking a weighted average across the four threads, Coupled mode requires 46.5 cycles per device evaluation. Although STS requires only 25 cycles per evaluation, the multiple threads of Coupled allows evaluations to overlap such that the aggregate running time is shorter (274 cycles versus 505 for STS). On single threaded code, Coupled and STS perform equally well; on threaded code, Coupled mode will execute in fewer cycles.

Restricting Communication

Data may need to be transferred from one cluster to another for several reasons. When two independent operations executing simultaneously on different clusters produce results needed by a subsequent operation, at least one of the results must be transferred. Thus data movement comes as a result of the compiler trying to exploit the maximum instruction level parallelism. Another source of data movement comes from operation results that must be used by multiple subsequent operations. One example is an eliminated common subexpression such as a redundant array index calculation. These values might be distributed to multiple clusters.

Since the number of buses and register input ports required to support fully connected function units is prohibitively expensive, some compromises must be made. Restricting communication between function units reduces cost without significantly affecting cycle count. The five different communication configurations that were simulated are described below:

1. Full: The function units are fully connected with no restrictions on the number of buses or register file write ports.

2. Tri-port: Each register file has three write ports. One port is used locally within a cluster by those units sharing the register file. The other two ports have their own buses and can be used by any function unit in another cluster.

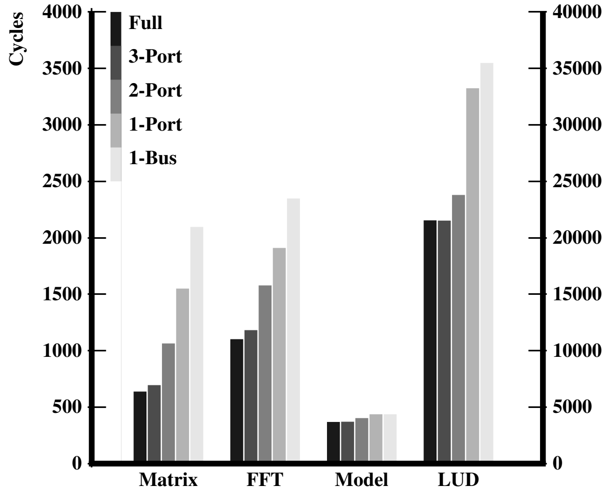

Figure 6: Cycle counts for restricted communication schemes of processor coupling on all benchmarks. **Matrix**, **FFT**, and **Model**, are referenced to the left axis while the scale for **LUD** is on the right. Cycle count increases dramatically when single buses and ports are used, but the Tri-Port scheme is nearly as effective as the fully connected configuration.

3. Dual-port: Each register file has two write ports. This is similar to Tri-Port with only one global register port.

4. Single-port: Each register file has a single write port with its own bus. Any function unit can use the port without interfering with writes to other register files.

5. Shared-bus: Each register file has two ports. One port is for use within a cluster while the other port is is connected to a globally shared bus. Arbitration is performed to decide which function unit may use the bus on a given cycle.

Figure 6 demonstrates how processor coupled performance is affected by restricting the amount of communication between function units. As expected, the number of cycles increases when function units must contend for buses and register ports.

Matrix, **FFT**, and **LUD** have high integer unit utilization because they calculate many common array indices, and are sharply affected when using a Single-Port or Single-Bus network. **Model** exhibits less instruction level parallelism, has low unit utilization, and is hardly affected by changing communication strategies.

Any restricted communication scheme trades chip area for increased cycle count. Tri-Port performs the best of the restricted configurations examined requiring an average of only 4% more cycles than the fully connected configuration. Tri-port can be implemented using only 2 global buses per cluster. The number of buses to implement a fully connected scheme, on the other hand, is proportional to the number of function units times the number of clusters. Furthermore, the completely connected configuration will require additional register ports. In a four cluster system the interconnection and register file area for Tri-Port is 28% that of complete connection.

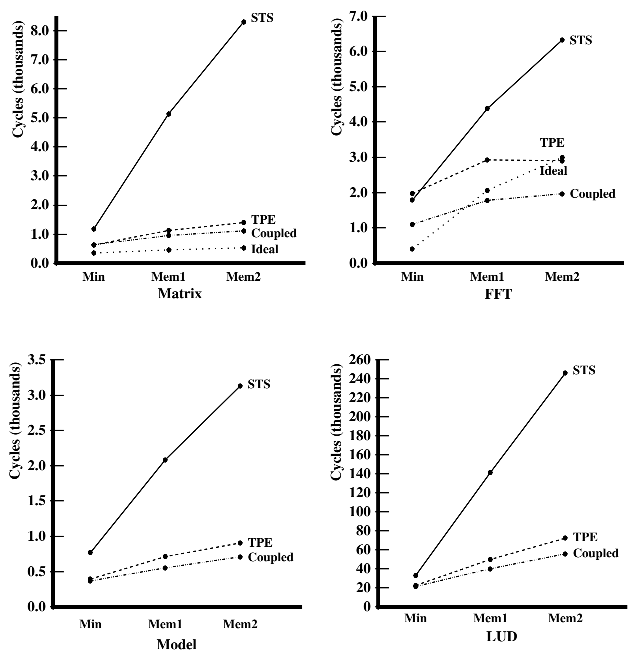

Figure 7: Cycle counts when memory latency is varied for all of the benchmarks. Increased memory latency affects the single threaded modes (STS and Ideal) more than the threaded modes (Coupled and TPE).

Variable Memory Latency

Long memory latencies due to synchronization or remote references degrade performance of any machine. Statically indeterminant memory latencies are particularly damaging for STS mode machines since long delays stall the entire computation. Multithreaded machines hide long memory latencies by executing other threads.

A five to ten percent miss rate is assumed for an on-chip cache depending on its size and organization. If a cache miss occurs, the memory reference must go off chip. When the requested data is in local memory, the reference might complete in 20 cycles. References to physically remote nodes can take 100 or more cycles. The three models of memory system performance used in simulations are:

- **Min**: single cycle latency for all memory references.
- **Mem1**: single cycle hit latency, 5% miss rate, and a miss penalty randomly distributed between 20 and 100 cycles.
- **Mem2**: similar to **Mem1** with a 10% miss rate.

Figure 7 shows how the cycle counts for different machine models are affected by long memory latencies. Since the compiler is able to use 490 registers in Ideal mode for **Matrix**, very few memory references need to be made. Thus long latencies hardly affect the ideal machine's cycle count. In **FFT**, however, the cycle count for Ideal mode increases dramatically because many loads and stores are required, and each delayed memory reference halts computation. Cycle count for STS mode rises in all benchmarks with increasing memory latency for similar reasons. Nearly 5.5 times as many cycles are needed on average for execution with **Mem2** parameters as with **Min** for STS.

The cycle count for Coupled does not increase as greatly in any of the benchmarks since other threads are executed when one thread is waiting for a long latency reference. On average, execution with **Mem2** parameters requires twice as many cycles as **Min**. If the compiler knew which references would cause long delays, it could create a schedule to try to mask long latencies. However, since memory latencies cannot be statically determined, runtime scheduling techniques, like those of processor coupling, must be used to mask the delay. Memory latencies can be quite long in a distributed memory parallel machine; masking of latency is a major advantage of Coupled over STS.

TPE is affected only a little more severely than Coupled by long memory latencies. Execution in **Mem2** mode requires 2.3 times as many cycles as **Min**. Like Coupled mode, TPE has other threads to run while waiting for long latency memory references. However, threads are allocated statically to specific clusters. If only one thread is resident on a cluster and it stalls waiting for a reference, the processor resources on that cluster go unused.

Number and Mix of Function Units

To determine the proper ratio between different types of units, we simulated all processor coupled machine configurations with up to four IUs and four FPUs while keeping the number of memory units constant at four. For our applications, simulation showed that a single branch unit is sufficient. Figure 8 displays the cycle counts for all the benchmarks as a function of the number of IUs and FPUs. The number of FPUs and IUs are on the X and Y axes. Cycle count is displayed on the Z axis.

The function unit requirements depend greatly upon the application. For **Matrix**, cycle count is highest when only one IU and one FPU are used, and decreases when more units are added. If the number of IUs is held constant and the number of FPUs is increased, the cycle count drops. The same holds true if the number of FPUs is constant and the number of IUs is varied. One FPU will saturate a single IU, but two IUs are needed to saturate a single FPU. Even though each benchmark consists primarily of floating point operations, this shows that integer units, which are used for synchronization and loop control, can also be a bottleneck. With a fixed number of function units, cycle count is minimized when the number of FPUs and IUs are equal.

The results for **FFT** are similar to those of **Matrix**. However, with four FPUs and one IU, the cycle count increases. This is due to additional IU operations required to move floating point array indices to remote memory units. Like **Matrix**, one FPU will saturate a single IU, but each additional IU improves performance. **LUD** shows much the same behavior as **FFT** with cycle count increasing as FPUs are added.

Model exhibits much less instruction level parallelism and does not benefit as greatly as the other benchmarks from additional function units. Cycle count is still minimized when four IUs and four FPUs are used.

For these benchmarks, the incremental benefit in cycles de-